1989

Handling Timing Constraints in Rapid Prototyping

Luqi

IEEE

Handling Timing Constraints in Rapid Prototyping

Luqi

Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

ABSTRACT

This paper presents the concepts and mechanisms for handling the real-time constraints of embedded software systems in rapid prototyping. Rapid prototyping of embedded systems can be accomplished using a Computer Aided Prototyping System (CAPS) and its associated Prototyping Language (PSDL). This system and language are used to aid the designer in the early stages of software engineering for hard real-time systems. Such systems contain time critical operations with maximum execution times, maximum response times, and minimum calling periods. Interrupt handling, synchronization, and periodic behavior are also described at a high level. Time critical operations in a real-time prototype are handled by constructing static and dynamic schedules to coordinate execution and to meet the required prototype execution times. The mechanisms for expressing timing constraints in PSDL are described along with their effects on the schedulers.

1. Introduction

Rapid prototyping can be used to reduce the risks of producing systems that do not meet user needs [10]. Traditional approaches to software development produce working code only near the end of the process. The goal of rapid prototyping is to develop an executable model of the intended system early in the development process. A prototype is an pilot version of a proposed system used as an aid to analysis and design. The code of a prototype usually cannot be used as the final implementation because it may not realize all aspects of the intended system. When utilized during the early stages of the development life cycle, rapid prototyping allows validation of the requirements, specification, and initial design, before valuable time and effort are expended on implementation software.

With the demand for hard real-time and embedded systems increasing and their particularly strict requirements on accuracy, safety and reliability, it is becoming critical that new approaches be proposed for the development of such systems since it can be very difficult to determine the timing constraints accurately or to understand their implications, even though they are the most important aspects of hard real-time systems [4, 13]. An embedded software system is part of some larger system, which it monitors and controls. The requirements of embedded systems typically have timing constraints for critical real-time control and high reliability. The feasibility of these systems depends on the ability to realize the timing constraints given in the requirements even under the worst possible operating conditions, because missing a timing constraint can cause catastrophic failures in such systems. Requirements for large embedded systems are difficult to meet without extensive prototyping.

Software engineers and end-users would benefit from an automated prototyping methodology for validating design specifications and functional requirements early in the development life cycle [13]. A fast, efficient, easy-to-use set of software tools would increase designer productivity and also allow the user to gain more confidence that the final software product is feasible. The computer aided prototyping system is a set of software tools which provides these capabilities [11]. It implements the rapid prototyping concept utilizing a high level prototyping language called PSDL [12]. This language is designed for prototyping hard real-time and large software systems and is based on the conceptual modeling of such systems.

This paper uses PSDL as a vehicle to analyze timing aspects of hard real-time systems and further describes the feasible paths to handling these hard real-time constraints by means of execution of real-time software prototypes. The approach described here makes practical validation of the timing constraints in the requirements of hard real-time systems via rapid prototyping possible. We concentrate on timing aspects here. Behavioral aspects of PSDL are described in [12].

Rapid prototyping initially establishes an iterative process between the user and the designer to concurrently define requirements and specifications for the critical aspects of the envisioned system. The designer then constructs a model or prototype of the system in PSDL. The prototype is a partial representation of the system, including only those critical attributes necessary for meeting user requirements, and is used as an aid in analysis and design rather than as production software. During demonstrations of the prototype, the user validates the prototype's actual behavior against its expected behavior. If the prototype fails to execute properly or to meet any critical timing constraints, the designer identifies required modifications and redefines the critical requirements and specifications based on feedback from the user (Fig. 1). This process continues until the user and the designer both agree that the prototype successfully meets the time critical aspects of the requirements. This iterative communication process should result in a model that ultimately meets the intended requirements of the user. Following this final validation, the designer uses the prototype as a basis for the design and eventual hand coding of the production software.

A good modeling strategy is important for making the rapid prototyping approach work for real-time systems. Systems can be designed and analyzed quickly only if they can be readily understood by the designers. This makes the ability to decompose large problems into independent subproblems essential. Timing constraints complicate the design of real-time systems because utilization of computing resources introduces interactions between otherwise unrelated parts of the system. A good modeling strategy should help to counteract this effect. This can be done in the following ways:

(1) by decoupling the behavioral aspects of a system from its timing properties to allow independent analysis of these two aspects, and

(2) by organizing timing constraints in a hierarchical fashion, to allow independent consideration of smaller subsets of timing constraints.

An effective modeling strategy should therefore support a set of abstractions useful for simplifying the timing aspects of systems with hard real-time constraints.

People are best at analyzing situations with a limited number of components and constraints, and tend to have difficulty in understanding nonlocal interactions in complex systems. Since shared computing resources introduce unavoidable global constraints on the set of time-critical computations in a real-time system, mechanical analysis of those constraints is desirable. This consideration motivates the need for a modeling strategy which allows the global timing constraints to be separated from the behavioral aspects of the system, and represented in a way that is amenable to mechanical processing. PSDL provides such a separation of timing and behavioral aspects, and supports computer-aided prototyping of both aspects.

[1]This research was supported in part by the National Science Foundation under grant number CCR-8710737.

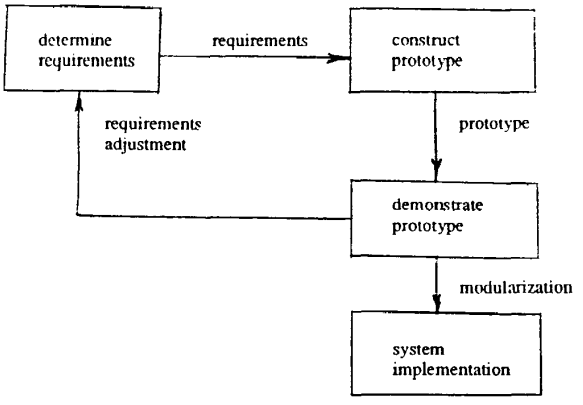


Fig. 1 Process of Requirements Determination and Validation

The modeling strategy used in the rapid prototyping of real-time systems should be integrated with a language supporting the modeling strategy and a systematic prototyping method for applying the strategy. A formal notation is essential for achieving a significant level of computer-aided design [3]. The prototyping language PSDL [12] supports a modeling strategy based on dataflow graphs augmented with non-procedural timing and control constraints. The language was designed together with the associated prototyping method [13] and the set of software tools in CAPS for supporting the method [11].

Section 2 discusses several approaches to modeling real-time systems. The real-time aspects of PSDL are described in section 3, and the associated software tools are described in section 4. Section 5 presents some conclusions.

2. Models for Real-Time Systems

The treatment of real-time constraints in this paper results from a fundamental analysis of hard real-time systems. Understanding the modeling strategies for such systems is essential for effectively handling timing constraints in the prototyping process. Methods for modeling real-time systems are essential for requirements analysis, specification, and design of systems with hard real-time constraints. A coherent framework for classifying real-time constraints can be used to organize complex sets of timing requirements and to guide the process of discovering the timing requirements associated with an embedded software system.

The framework presented in [4] classifies timing requirements as maximum or minimum constraints on the lengths of time intervals between pairs of events, where the endpoints of the intervals are classified as one of the following types: (1) stimulus-stimulus, (2) stimulus-response, (3) response-stimulus, and (4) response-response. A third kind of timing constraint identified in [4] is the durational constraint, which states that an event must occur for a time interval of the specified length. This framework treats events as signals rather than as instants of time, and tries to treat both digital and analog systems in the same terms. In particular, events are not required to be instantaneous, although events without durational constraints are instantaneous by default. This feature introduces an ambiguity in the definitions of the four kinds of time intervals, because it is not specified whether the intervals are delimited by the beginning or the ends of the events forming the endpoints of the intervals. It also introduces an artificial distinction between durations of signals and other kinds of time intervals.

Another weakness of this framework is the implicit assumption that every timing constraint involves just a single pair of events. This implicit assumption is undesirable because it does not allow simple descriptions of periodic processes, which are an important aspect of real-time systems.

A different framework for describing real-time constraints is provided by RTL (Real Time Logic) [7]. RTL makes a distinction between actions and events. Actions require a bounded non-zero amount of system resources, such as CPU time. Events are instantaneous, requiring no system resources and serving only as temporal markers. Four kinds of events are distinguished in RTL for hard real-time systems:

(1) External events, such as an operator pushing a button.

(2) Start events marking the initiation of an action.

(3) Stop events marking the completion of an action.

(4) State variable transition events marking a change in some attribute of the system.

The durational events of [4] can be expressed in RTL in terms of the duration of an interval delimited by two state variable transition events, marking the appearance and disappearance of the signal. RTL also provides a way to index all of the occurrences of each type of event in the order in which they occur, via the occurrence function "@". This facility is important because it allows the description of periodic processes. RTL is a powerful notation which can be used to mechanically check whether a given safety assertion follows from RTL system specifications via a decision procedure for Presburger arithmetic [7]. The primitives of RTL are very simple and provide the ability to describe timing constraints in detail. RTL is a good choice for defining the semantics of notations at higher levels of abstraction, because a scheme for translating a notation into RTL allows the application of the RTL decision procedure to the other notations.

Spec, a powerful specification language for large and real-time systems based on quantifiers and predicate logic, uses another framework for timing constraints called the event model of computation [1]. In the event model, computations are described in terms of modules, messages, and events. A module is a black box that interacts with other modules only by sending and receiving messages. A message is a data packet that is sent from one module to another. An event occurs when a message is received by a module at a particular instant of time. The events at a module occur one at a time in a well defined order, while events at different modules need not be ordered.

Modules can be used to model external systems such as users and peripheral hardware devices, as well as software components. Modules are active black boxes, which have no visible internal structure. The behavior of a module is specified by describing its interface. The interface of a module consists of the set of stimuli it recognizes and the associated responses. A stimulus is an event, and the response is the set of events directly triggered by the stimulus. The events in the response consist of the arrivals of the messages sent out by the module because of the stimulus.

As in RTL, events in the Spec event model are instantaneous and serve to define points in time. Every event is associated with the instant of time when an incoming message crosses the boundary of a module. An event marks the time when a module accepts a message and starts the computation of the module's response. The Spec event model does not distinguish a completion event for an action, since the events in which the messages sent out in response to an event arrive at their destinations serve this purpose. The event model treats the environment of a system as a set of modules, so that there is no formal distinction between internal and external events, as is the case in RTL. In the Spec event model all state changes are instantaneous and localized to some module. The time of the state change is taken to be the same as the time of the event at which the message triggering the state change arrived at the module containing the state. An implementation may take a non-zero time interval to realize a state change, but all state changes triggered by an event at a module must be complete before the module can accept another message. Since all interactions between two modules occur via the transmission of messages, the exact instant at which a state change takes place cannot affect the observable behavior of the system until the module accepts its next message. Since the delays between events and the completion of the state changes they trigger are not externally observable, they are abstracted out in the Spec model. Spec events thus unify the four types of events distinguished in RTL.

Spec allows the use of arbitrary predicates in first order logic to express constraints on the delay and period associated with each type of event. The *delay* is the interval between an event representing a stimulus to a module and another event representing a response of the module to that stimulus. Constraints on the delay correspond to the stimulus-response and response-stimulus constraints of [4]. The *period* is the interval between two consecutive events of the same type at the same module. Constraints on the period correspond to the stimulus-stimulus and response-response constraints of [4].

The Spec language supports the definition of temporal events as well as events triggered by external stimuli. Each module can send messages to itself at absolute times determined by its local clock. The arrival of such a message is called a *temporal event*. Temporal events allow modules to initiate actions in addition to just responding to external stimuli.

The time at which a temporal event occurs can be constrained by an arbitrary predicate in first order logic. Such predicates can easily describe periodic processes, the most common kind of temporal events, as well as other kinds of temporal events. Like RTL, Spec provides primitives for defining timing constraints at a detailed level. Spec also has facilities allowing users to define high level timing constraints, but high level abstractions for timing are not built into the language or pre-defined.

The modeling method used in PSDL, which is the basis for the approach to handling timing constraints presented in this paper, is described in the next section.

3. Timing Constraints in Rapid Prototyping

PSDL is a language designed for clarifying the requirements of complex real-time systems, and for determining properties of proposed designs for such systems by means of prototype execution. The language was designed to simplify the description of such systems [12] and to support a prototyping method that relies on a novel decomposition criterion [13]. PSDL is also the basis for a computer-aided prototyping system [11] that speeds up the prototyping process by exploiting reusable software components [10] and providing execution support [9, 15, 18] for high level constructs appropriate for describing large real-time systems in terms of an appropriate set of abstractions [2]. The execution support system contains a static scheduler for scheduling operators with hard real-time constraints, a dynamic scheduler for scheduling operators without timing constraints, and a translator for generating executable code.

An important goal of PSDL is to simplify the design of systems with hard real-time constraints. The need for meeting real-time deadlines often results in designs where code for conceptually unrelated tasks must be interleaved, complicating the design of such systems, and making their implementations hard to understand [6]. PSDL handles this problem by presenting a high-level description in terms of networks of independent operators, and allowing the interleaving of the code to be handled by an automatic translator that generates lower level code. High-level synchronization is handled by using dataflow streams to coordinate the arrival of corresponding data values from different sources. Static scheduling of time-critical operators [8] eliminates the need for other kinds of explicit synchronization by the system designer. Low-level synchronization primitives are needed in the implementation of PSDL only for ensuring that the read and write operations on data streams are performed as atomic operations. Since those primitives are needed in just one small and simple part of the code in the PSDL execution support system, PSDL can be effectively supported by any of the common mutual exclusion mechanisms provided by operating systems.

The PSDL language is based on a computation model which treats software systems as networks of operators communicating via data streams. The computational model is an augmented directed graph

$$G = (V, E, T(v), C(v))$$

where V is the set of vertices, E is the set of edges, T(v) is the set of timing constraints for each vertex v, and C(v) is the set of control constraints for each vertex v. Each vertex is an **operator** and each edge is a **data path.** Each of the four components of the graph are described in more detail below. The semantics of a PSDL system description is determined by the associated augmented graph and the semantics of the operators appearing in the diagram.

All PSDL operators are state machines. Some PSDL operators are functions, i.e. machines with only one state. When an operator fires, it reads one data value from each of its input streams, undergoes a state transition, and writes at most one data value into each of its output streams. The output values can depend only on the current set of input values and the current state of the operator. State transitions and input/output operations on data streams can occur only when the associated operator fires. The firing of an operator is controlled by the associated timing and control constraints. Operators can be triggered by the arrival of a set of input data values or by a periodic temporal event.

PSDL operators communicate by means of named **data streams**. All PSDL data streams can carry both normal data values and tokens representing exceptions. All of the normal data values carried by a stream must be instances of a specified abstract data type associated with the stream.

There are two different kinds of data streams in PSDL, **dataflow** streams and **sampled** streams. Dataflow streams are used in applications where the values in the stream must not be lost or replicated and the firing rates of the producers and consumer are the same, while sampled streams are used in applications where a value must be available at all times and values can be replicated without affecting their meaning.

Any PSDL operator can have timing constraints associated with it. An operator is **time-critical** if it has at least one timing constraint associated with it, and is **non time-critical** otherwise. The timing constraints together with the control constraints determine when the operator can be fired, and when it must be fired. All PSDL timing constraints can be represented by constants denoting lengths of time intervals. There are several different kinds of timing constraints, which can be classified into those that apply to all time-critical operators, those that apply only to operators triggered by periodic temporal events, and those that apply only to operators triggered by the arrival of new data. Temporal events occur at specified sets of absolute times.

Every time-critical operator must have a **maximum execution time** (MET) to allow the construction of a static schedule. The MET of an operator is an upper bound on the length of the **execution interval** (EI) for the operator. All of the actions that may be required to fire an operator once must fit into the execution interval. These actions are listed below.

(1) Reading values from input data streams.

(2) Evaluating triggering conditions.

(3) Calculating output values.

(4) Evaluating output guards.

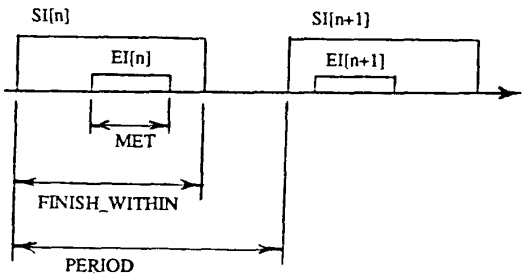
(5) Writing values into output streams.

The execution interval for an operator does not include scheduling delays. A scheduling delay is the time between the writing of a value into a data stream by a producer operator and the reading of that value by the consumer operator.

Operators triggered by temporal events are periodic in PSDL. Every periodic operator must have a **period** (PERIOD) and may have a **deadline** (FINISH_WITHIN). These two timing constraints partially determine the set of **scheduling intervals** (SI) for the operator. Each periodic operator must be fired exactly once in each scheduling interval, and must complete execution before the end of the scheduling interval. The period is the length of time between the start of any scheduling interval and the start of the next scheduling interval. The deadline is the length of each scheduling interval. The starting time of the first scheduling interval for each operator is determined by the static scheduler, subject to the scheduling constraints described below.

There is an implicit constraint on the order in which PSDL operators can be scheduled to fire, known as the dataflow precedence constraint. The dataflow precedence constraint requires the initial firings of all operators with timing constraints to occur in an order consistent with the **dataflow ordering**, which is defined in terms of the PSDL computational model as follows. Construct the **precedence graph** by taking all of the nodes in the augmented graph G defined above, and taking only the edges of G that do not have explicitly declared initial values. Since any cycle of G must contain at least one edge with a declared initial value in a well formed PSDL program, the precedence graph is acyclic. The dataflow ordering is the transitive closure of the precedence graph, which results in a strict partial ordering. For any pair of operators (**a**, **b**), if **a** precedes **b** in the dataflow ordering then **a** must be scheduled to fire before **b** is scheduled to fire for the first time.

The relation between the timing constraints, scheduling intervals, and execution intervals for a periodic operator is illustrated in Fig. 2. The execution intervals and scheduling intervals in the diagram are indexed by integers in the order of their occurrence. Thus SI[n] denotes the n-th scheduling interval for the operator and EI[n] denotes the n-th execution interval for the operator. The static scheduler takes the length of each execution interval to be equal to the maximum execution time to allow for worst case conditions. If a time-critical operator completes before the end of the execution interval reserved for it by the static scheduler, the remaining time in the execution interval is used by the dynamic scheduler for the



SI[n] = n-th scheduling interval
EI[n] = n-th execution interval

Fig. 2 Timing Constraints for a Periodic Operator

execution of a non time-critical operator.

Since each execution interval must be a subset of the corresponding scheduling interval, every well-formed periodic operator must satisfy the constraint

MET <= FINISH_WITHIN.

The degree of freedom enjoyed by the static scheduler is characterized by the slack, which is defined by

slack = FINISH_WITHIN - MET.

The length of time between the start of an execution interval and the start of the next execution interval can vary between

PERIOD - slack

and

PERIOD + slack,

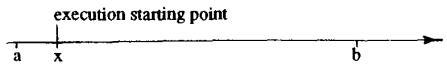
as illustrated in Fig. 3. If FINISH_WITHIN is not specified explicitly then

FINISH_WITHIN = MET,
slack = 0,

and the time between the starting points of each pair of consecutive execution intervals is exactly equal to the period.

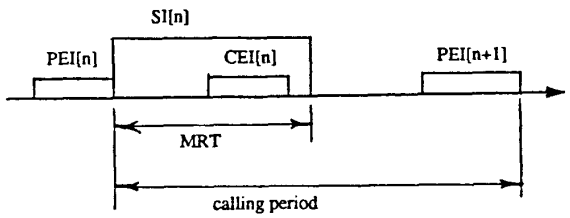
Operators triggered by the arrival of new data values are sporadic. Timing constraints for sporadic operators are optional. Sporadic operators with timing constraints must have both a **maximum response time** (MRT) and a **minimum calling period** (MCP) in addition to an MET. The MRT is an upper bound on the **response time**, while the MCP is a lower bound on the **calling period**. The relation between these quantities is illustrated in Fig. 4. SI[n] denotes the n-th scheduling interval for the consumer operator, which is sporadic and time-critical. CEI[n] denotes the n-th execution interval for the consumer operator, and PEI[n] denotes the n-th execution interval for the producer operator, which is assumed here to be time-critical also. The response time associated with a consumer operator is measured from the end of the execution interval for the producer operator of the triggering data value to the end of the execution interval for the consumer operator of the triggering data value.

Unlike the MET, the MRT includes a scheduling delay. The MRT gives the length of the scheduling interval. The static scheduler may not be able to use the entire scheduling interval if the producer is non time-critical, because the ending time of the producer's execution interval is not known to the static scheduler in that case.



slack = FINISH_WITHIN - MET
a = period - slack
b = period + slack
a <= x <= b

Fig. 3 Scheduling Freedom for a Periodic Operator



PEI[n] = n-th producer execution interval
CEI[n] = n-th consumer execution interval
SI[n] = n-th scheduling interval

Fig. 4 Timing Constraints for a Sporadic Operator

The calling period of an operator is the length of time between the end of the execution interval for the producer of the triggering data value and the end of the execution interval for the producer of the next triggering data value. The calling period must not be less than the MCP. The MCP of an operator constrains the behavior of the producers of the triggering data values rather than constraining the behavior of the operator itself. An MCP constraint is needed to allow the realization of a maximum response time constraint with a fixed amount of computational resources, via a limit on the frequency with which new data can arrive. Violation of an MCP constraint should result in a warning message. The absence of such violations can be guaranteed a priori only if the producer of each triggering data value is scheduled statically. An important case where such a guarantee is not possible occurs when the producer of the triggering data value is an asynchronous active sensor controlled by factors external to the system. In such cases data may be lost if it arrives too frequently.

PSDL provides a simple framework for describing real-time systems at a level convenient both for the designer of a prototype and for a set of software tools. The apparent simplicity of the designer's view stems from the following properties.

(1) *Locality of information.* There is no hidden data coupling between operators due to nonlocal data references and no hidden control coupling between operators due to interrupts or exceptions.

(2) *Locality of timing constraints.* All PSDL timing constraints are associated with individual operators, and are defined in terms or events and intervals that can be defined in terms of a black-box view of the operator. The small number of concepts involved induces a simple and regular structure on the timing constraints of the system which allows the constraints to be organized in a coherent fashion amenable to localized understanding and analysis.

(3) *Hierarchical Organization.* The timing constraints in PSDL are organized hierarchically as well as the data and control aspects of the system. This allows the lower level timing constraints used to implement an operator at a higher level of abstraction to be ignored when analyzing the interactions of the abstract operator with its neighbors in the high level data flow network. This allows the timing constraints in the system to be partitioned into independent units that can be designed and analyzed in isolation from each other.

Despite their simplicity, the primitives of PSDL are well suited to modeling practical real-time systems. Both periodic and data driven processes are describable directly, by defining either a period or a data trigger for the corresponding operator. Synchronous interactions between operators or system components are modeled as dataflow streams while asynchronous interactions are modeled by sampled streams. Systems containing both synchronous and asynchronous components can be described easily and safely, since the stream types are automatically derived from the context in which the stream is used, and need not be explicitly considered by the designer.

The timing constraints of PSDL are also amenable to automatic analysis and realization, as described below.

4. Computer-Aided Handling of Timing Constraints

The rapid construction of a prototype in PSDL is made possible by

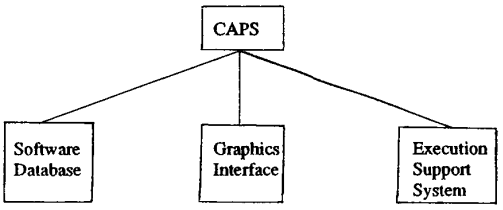


Fig. 5 Major Software Tools of CAPS

the associated prototyping method and support environment. The support environment reduces the efforts of the designer by automating some of the tasks involved in prototype construction.

The Computer Aided Prototyping System (CAPS) relies on three major software tools, as illustrated in Fig. 5, to assist the designer in constructing and executing a PSDL prototype. First, the computer-aided environment includes a software base management system which creates uniform retrieval specifications for Ada software modules in the software database and later retrieves these reusable modules for assembling the executable prototype. Second, a graphics-capable user interface including a syntax-directed editor expedites the designer's data entry at a terminal and prevents syntax errors in the design. Finally, an execution support system demonstrates and measures the prototype's performance and analyzes the accuracy of the design specifications.

The execution of PSDL prototype models is made possible through the use of an automated Execution Support System which consists of three parts, the translator, the static scheduler, and the dynamic scheduler. We will first examine the translator [15]. Its basic function is to convert the prototyping model input by the designer in the form of PSDL source code into Ada source code. Output from the translator is provided to the Ada compiler/linker along with some additional information from the static scheduler to produce Ada object code. The object code is then exported to the operating system and can be run for test and demonstration purposes. The translator creates code to implement the PSDL operators as procedures which will be called by the main subprogram/schedule created by the static scheduler. The translator does not generate any code from PSDL real-time constraints.

The static scheduler subsystem of the execution support system represents the single most important component of CAPS for meeting the basic requirement for computer-aided rapid prototyping of hard real-time systems [9]. The static scheduler specifically addresses only those PSDL operators with critical timing constraints. The performance of these operators determines whether the system, as designed, will meet the required timing specifications. The primary purpose of the static scheduler is the creation of a static schedule which gives the precise execution order and timing of PSDL component operators with hard real-time constraints in such a manner that all timing constraints are guaranteed to be met [14]. Provided that such a schedule is feasible given the system specifications, the static schedule contains the preallocated starting time and execution time for each critical operator. This structure implicitly denotes the precedence relationships between the operators. Rapid prototyping in general would benefit from CAPS without a static scheduler, but such a facility is needed to realize systems with hard real-time constraints. The static scheduler contributes to increased gains in designer productivity and system reliability in the development of hard real-time systems since the construction and validation of a static schedule is a labor intensive process that cannot be carried out both rapidly and reliably without the benefit of automated software tools.

The third component of the execution support system is the dynamic scheduler. This scheduler operates at run-time along with the prototype model and is designed to control the execution of all non-critical operators in the program. A non-critical operator is one which is not subject to hard real-time constraints. The dynamic scheduler is invoked whenever there is spare time in the static run-time schedule created by the static scheduler. Progress of non-critical operators occurs at unpredictable rates, which implies that non-critical operators should be connected to critical operators using sampled streams rather than data flow streams. At that time the dynamic scheduler commences execution of the next available module in its set of operators and continues to invoke non-critical modules until the available time is exhausted. At that point, operation of the dynamic scheduler is interrupted and control is returned to the static scheduler to continue the time critical operations. This process is simplified by the locality of data references in PSDL, since this property allows the execution of an operator to be interrupted at any time except during read or write operations on data streams. Future enhancements identified in the current design of the dynamic scheduler would provide debugging facilities and statistical information [5].

The remainder of this section contains implementation guidelines describing how the static scheduler functions [9, 18]. The design described in this paper addresses a single processor implementation only, and the modifications to this design needed to utilize multi-processors and concurrent processing are not addressed explicitly.

The initial input to the static scheduler is a text file containing the PSDL prototype program created jointly by the designer and user. An intermediate output to the dynamic scheduler is a file containing the non-time-critical operators in the PSDL program. The final output of the static scheduler to the compiler/linker/exporter is an Ada source file containing the static schedule. The compiler/linker/exporter compiles and links this program together with the compiled program produced by the translator. This combined program is the executable Ada program used by the dynamic scheduler to demonstrate the prototype's performance. The data flow diagram shown in Fig. 6 illustrates the conceptual design of the static scheduler and outlines its five major functions [18]. The following subsections describe each of these major functions.

4.1. Read_PSDL

Following initiation by the dynamic scheduler, the static scheduler's first major function is reading and processing the PSDL prototype program. The static scheduler requires only the information which identifies critical operators along with their timing constraints and the link statements which syntactically describe PSDL implementation graphs. A special purpose translator identifies and extracts this information from the PSDL source program. This process creates a file containing operator identifiers, timing information, and link statements.

4.2. Pre-Process File

The static scheduler's next major function is classifying the contents of this file and performing basic validity checks. The input file contains all of information extracted from the PSDL program. This information must be divided into three separate files based on its destination and the additional processing required. The Non-Crits file contains a list of all non-time-critical operators for the dynamic scheduler. The Operator file contains all time-critical operators and their associated timing constraints. The Links file contains the link statements which syntactically describe the PSDL implementation graphs.

Validity checks are performed on the timing information contained in the Operator file to increase the probability of successfully creating a feasible schedule. At a minimum three validity checks are performed at this

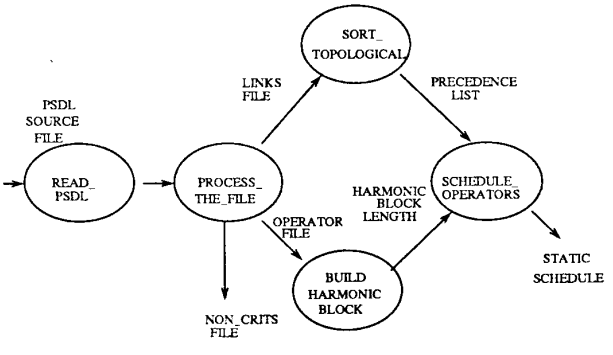


Fig. 6 DFD for the Static Scheduler

stage. The Operator file contains operator names, maximum execution times (MET), maximum response times (MRT), minimum calling periods (MCP), fixed periods, and deadlines (FINISH_WITHIN). The first check verifies that each operator with a MCP also has a MRT, and if it is missing it is calculated as (MRT = FINISH_WITHIN) or (MRT = MET). The second check verifies that (MET <= period) for operators with fixed periods. This constraint is imposed by the assumption that only a single processor is available. The third check verifies that (MET <= MRT) [17]. Failure of any of these checks causes an error message to be submitted to the user interface.

4.3. Sort_Topological

The static scheduler's next major function is to perform a topological sort on the link statements in the Links file. The requirement for a topological sort implies that operators producing each data stream must be scheduled before the operators consuming that data stream. The output of this process is a precedence list of the time-critical operators defining the exact order in which they will be executed. Fig. 7 shows an augmented acyclic digraph and the corresponding topologically sorted sequence of link statements. In this type of digraph the order of the nodes is not completely determined. In the example the decision to choose the "a.A" link first and the "d.A" link last was made arbitrarily.

4.4. Build_Harmonic_Blocks

The second output of the "Pre-Process_File" function, the Operator file, is the input to the "Build_Harmonic_Blocks" function. A harmonic block is a set of periodic operators, where the periods of all the operators in the set are exact multiples of a calculated base period [14]. This design approach treats each harmonic block as an independent scheduling problem, which requires one processor for each harmonic block. Our approach utilizes the capabilities for concurrency and multi-processing which are normally a requirement for complex, hard real-time systems. The implementation described in this paper addresses a single processor environment only, and the procedures described for generating the final static schedule assume that only one harmonic block is created. The rest of this section describes how sporadic operators are converted to their periodic equivalents, how base periods and block periods are calculated, and how harmonic blocks are created.

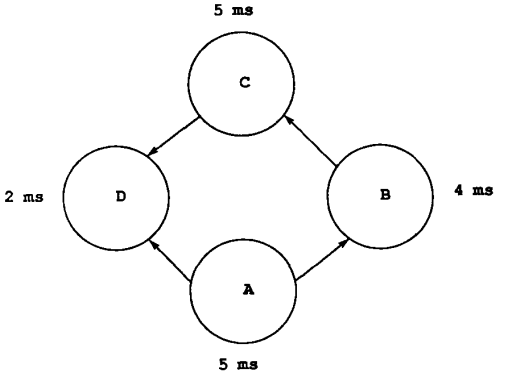


Fig. 7. PSDL Augmented Acyclic Digraph

The Operator file contains all the time-critical operators from the PSDL program together with their timing constraints. This includes both periodic and sporadic operators. Periodic operators are triggered at regular intervals, while sporadic operators are triggered by the arrival of new data. In order to guarantee the execution of sporadic operators within the MRT, it is necessary to assume new data cannot arrive more often than once per the MCP. To enable the execution of all operators in a predictable manner that can be described by a static schedule, sporadic operators are implemented by their calculated periodic equivalents [14]. The calculation of an equivalent period requires that all sporadic operators have values for MCP, MRT, and MET satisfying the following relationships.

1. MET <= MRT
2. MET <= MCP

The first condition ensures (MRT - MET) is positive, while the second allows a single processor implementation. The equivalent period is given by [14]

P = minimum(MCP, MRT - MET).

A single processor implementation also requires (MET <= P) to allow the operator to complete within the calculated period.

In the single processor case, all operators are allocated to the same harmonic block. In the multiple processor case, blocks are created one at a time by collecting all operators whose periods are exact multiples of the minimum period for all the unallocated operators. The base period of a block is the GCD (greatest common divisor) of all periods in the block, while the block period is the LCM (least common multiple) of all the periods in the block. The base period is used to relieve scheduling congestion via a second pass which moves operators to the block with the longest base period that evenly divides the operator period. The block period defines the length of the static schedule for the block.

4.5. Schedule_Operators

The "Schedule_Operators" function uses the "Precedence_List" and "Harmonic_Block" files, which are generated by the "Sort_Topological" and "Build_harmonic_Blocks" functions, respectively. The Precedence_List file defines the required execution order, while the Harmonic_Block file defines the set of operators to be scheduled and the length of the static schedule. The resulting static schedule is a linear table giving the exact execution start time for each time-critical operator and the reserved maximum execution time (MET) within which the operator must complete execution.

The algorithm used in this implementation is a two step process. The first step allocates the initial execution interval for each operator [16]. in the order given by the Precedence_List, using the relation

interval = (current_time, current_time + MET)

where the current_time is the beginning of the currently unallocated time interval in the block period. This step also creates a firing interval for each operator, during which the second step must schedule the next execution of the operator. The firing interval gives the lower and upper bound for the next possible starting time of the operator. For example, OP_2 in Fig. 8 is scheduled to start execution at time 2 and to complete by time 3, based on its MET of 1. Since OP_2 has a period of 10, it cannot fire again before time 12, the lower bound for its firing interval. OP_2 must fire by time 21, the upper bound, in order to ensure completion by the end of the second period at time 22.

The second step schedules subsequent firings of the operators, which are allocated in earliest-lower-bound-first order. In the example of Fig. 8 the operators are scheduled in the order [OP_1, OP_2, OP_4] during the first iteration of the second step. Since the period of OP_3 (20) is the same as the block period, it is scheduled to fire only once in the static schedule. As each operator is scheduled, this process verifies that both

1. current_time + MET <= block period, and
2. current_time <= upper bound.

Assume given

PRECEDENCE_LISTS (OP_1, OP_2, OP_3, OP_4)
HARMONIC_BLOCK_LENGTH = 20

and

OPERATOR_ID	MET	PERIOD
OP_1	2	10
OP_2	1	10
OP_3	3	20
OP_4	1	10

result

STATIC SCHEDULE

OP_ID	START TIME	END TIME	FIRING INTERVAL
OP_1	0	2	(10,18)
OP_2	2	3	(12,21)
OP_3	3	6	(23,40)
OP_4	6	7	(16,25)

and

HARMONIC BLOCK

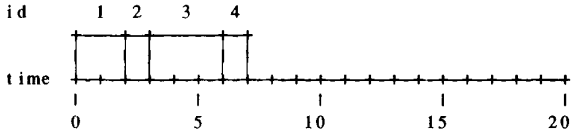


Fig. 8 Static Schedule after First Process

STATIC SCHEDULE

OP_ID	START TIME	END TIME	FIRING INTERVAL
OP_1	0	2	(10,18)
OP_2	2	3	(12,21)
OP_3	3	6	(23,40)
OP_4	6	7	(16,25)
OP_1	10	12	(20,28)
OP_2	12	13	(22,31)
OP_4	16	17	(26,35)
OP_1	20	22	(30,38)
OP_2	22	23	(32,41)
OP_3	23	26	(43,60)
OP_4	26	27	(36,45)
OP_1	30	32	(40,48)
OP_2	32	33	(42,51)
OP_4	36	37	(46,55)

and

HARMONIC BLOCK

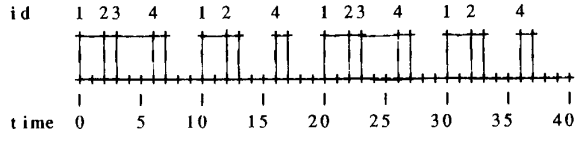


Fig. 9 Static Schedule for 2 Block Periods

Failure to meet either condition results in an infeasible schedule, resulting in an exception and an appropriate error message at the user interface. These checks ensure that the process produces a feasible schedule whenever it terminates without an exception.

The process also calculates new firing intervals for each process scheduled. Fig. 9 shows the static schedule after three iterations of the process, along with a timing chart for two block periods. This figure illustrates the importance of a robust static schedule. Once all operators are correctly scheduled for an entire block period, all subsequent schedules are delayed copies of the first, hence the term "static schedule". In the example shown, the static schedule is complete after only one iteration of the second step, since at that point the lower bounds of all the firing intervals are greater than or equal to the block period.

5. Conclusions

The mechanism discussed in this paper for handling timing constraints in rapid prototyping via PSDL offers the potential of serving as a valuable software development tool specifically for hard real-time systems. The prototyping process is speeded up by automation and by reducing the conceptual burden of the analyst and prototype designer. PSDL has constructs for recording timing constraints, defining operator and data abstractions, and for specifying non-procedural control constraints. The most important aspects of real-time system design are enforcing maximum execution time, maximum response time, minimum calling period, and data synchronization constraints. The details of how to handle these characteristics of hard real-time systems are discussed in this paper. The execution support system of CAPS described in section 4 allows the designer to check the feasibility of a set of real-time constraints by monitoring and evaluating the execution of the prototype model. PSDL offers a software tool that is a practical way to support rapid prototyping of hard real-time software systems. We can validate hard real-time constraints by executing the constructed prototype.

Current research projects to conceptualize components of the CAPS Execution Support System are reflected by the discussion of handling timing constraints in rapid prototyping contained in this paper. These efforts empirically demonstrate that the initial goal of providing an automated execution environment for hard real-time software design and specification prototypes is feasible. CAPS will provide software designers with an automated tool allowing validation of prototypes for hard real-time or embedded systems before extensive time and money are invested in production software. Additional research projects are currently underway to conceptualize, implement, and integrate the handling of timing constraints in rapid prototyping and make the computer-aided design of hard real-time systems feasible, practical, and reliable.

1. V. Berzins and Luqi, *Software Engineering with Abstractions: An Integrated Approach to Software Development using Ada*, Addison-Wesley, 1988.
2. V. Berzins, "The Design of Software Interfaces in Spec", in *to appear in Proceedings of the International Conference on Computer Languages*, Miami, Oct. 1988.
3. V. Berzins and Luqi, "Languages for Specification, Design and Prototyping", in *Handbook of Computer-Aided Software Engineering*, Van Nostrand Reinhold, 1988.
4. M. Chandrasekharan, B. Dasarathy and Z. Kishimoto, "Requirements-Based Testing of Real-Time Systems: Modeling for Testability", *IEEE Computer 18*, 4 (Apr. 1985), 71-80.
5. S. Eaton, "An Implementation Design of A Dynamic Scheduler for a Computer Aided Prototyping System", M. S. Thesis, Computer Science, Naval Postgraduate School, Monterey, CA, March 1988.
6. S. Faulk and D. Parnas, "On Synchronization in Hard-Real-Time Systems", *Comm. of the ACM 31*, 3 (Mar. 1988), 274-187.
7. F. Jahanian and A. Mok, "Safety Analysis of Timing Properties in Real-Time Systems", *IEEE Trans. on Software Eng. SE-12*, 9 (Sep. 1986), 890-904.
8. D. Janson and Luqi, "A Static Scheduler for the Computer Aided Prototyping System", *in Proceedings of COMPASS 88*, Gaithersburg, MD, June 1988.

9. D. Janson, "A static Scheduler for Hard Real-Time Constraints in the Computer Aided Prototyping System", M. S. Thesis, Computer Science, Naval Postgraduate School, Monterey, CA, March 1988.

10. Luqi, "Rapid Prototyping for Large Software System Design", Ph. D. Thesis, University of Minnesota, 1986.

11. Luqi and M. Ketabchi, "A Computer Aided Prototyping System", *IEEE Software 5*, 2 (March 1988), 66-72.

12. Luqi, V. Berzins and R. Yeh, "A Prototyping Language for Real-Time Software", *IEEE Trans. on Software Eng.*, October, 1988.

13. Luqi and V. Berzins, "Rapidly Prototyping Real-Time Systems", *IEEE Software*, Sep. 1988, 25-36.

14. Luqi and V. Berzins, "Execution of a High Level Real-Time Language", in *Proc. of the Real-Time Systems Symposium*, Dec. 1988.

15. C. Moffitt, "Development of a Language Translator for a Computer Aided Prototyping System", M. S. Thesis, Computer Science, Naval Postgraduate School, Monterey, CA, March 1988.

16. A. K. Mok, "The Decomposition of Real-Time System Requirements into Process Models", *IEEE Proc. of the 1984 Real Time Systems Symposium*, Dec. 1984, 125-133.

17. A. K. Mok, *The Design of Real-Time Programming Systems Based on Process Models*, IEEE, 1984.

18. J. O'Hern, "A Conceptual Design of a Static Scheduler for Hard Real-Time Systems", M. S. Thesis, Computer Science, Naval Postgraduate School, Monterey, CA, March 1988.